A gentle introduction to the Blockchain and Smart contracts

Giovanni Ciatto { giovanni.ciatto@unibo.it }

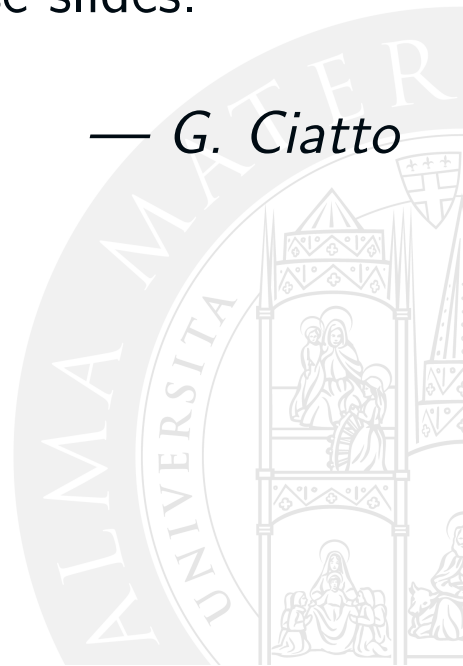
Talk @ Autonomous Systems Course, A.Y. 17/18
Dipartimento di Informatica, Scienza e Ingegneria—Università di Bologna

May 30, 2018

Acknowledgements

I wish to thank my supervisor Prof. Andrea Omicini, and my colleagues Prof. Enrico Denti, Dr. Stefano Mariani, and Dr. Roberta Calegari for the many fruitful discussions which I tried to synthesise in these slides.

— *G. Ciatto*



Talk Outline

1. State Machine Replication
2. The blockchain's main elements
3. Smart contracts
4. Research perspectives

State Machine Replication (SMR) [24, 10]

Main idea

Executing the same (not necessarily finite) state machine[a] over multiple independent (possibly distributed) processors, in parallel, in order to achieve:

- fault tolerance (to stops, crashes, lies, bugs, etc)
- availability and reactivity
- data / software replication & untamperability

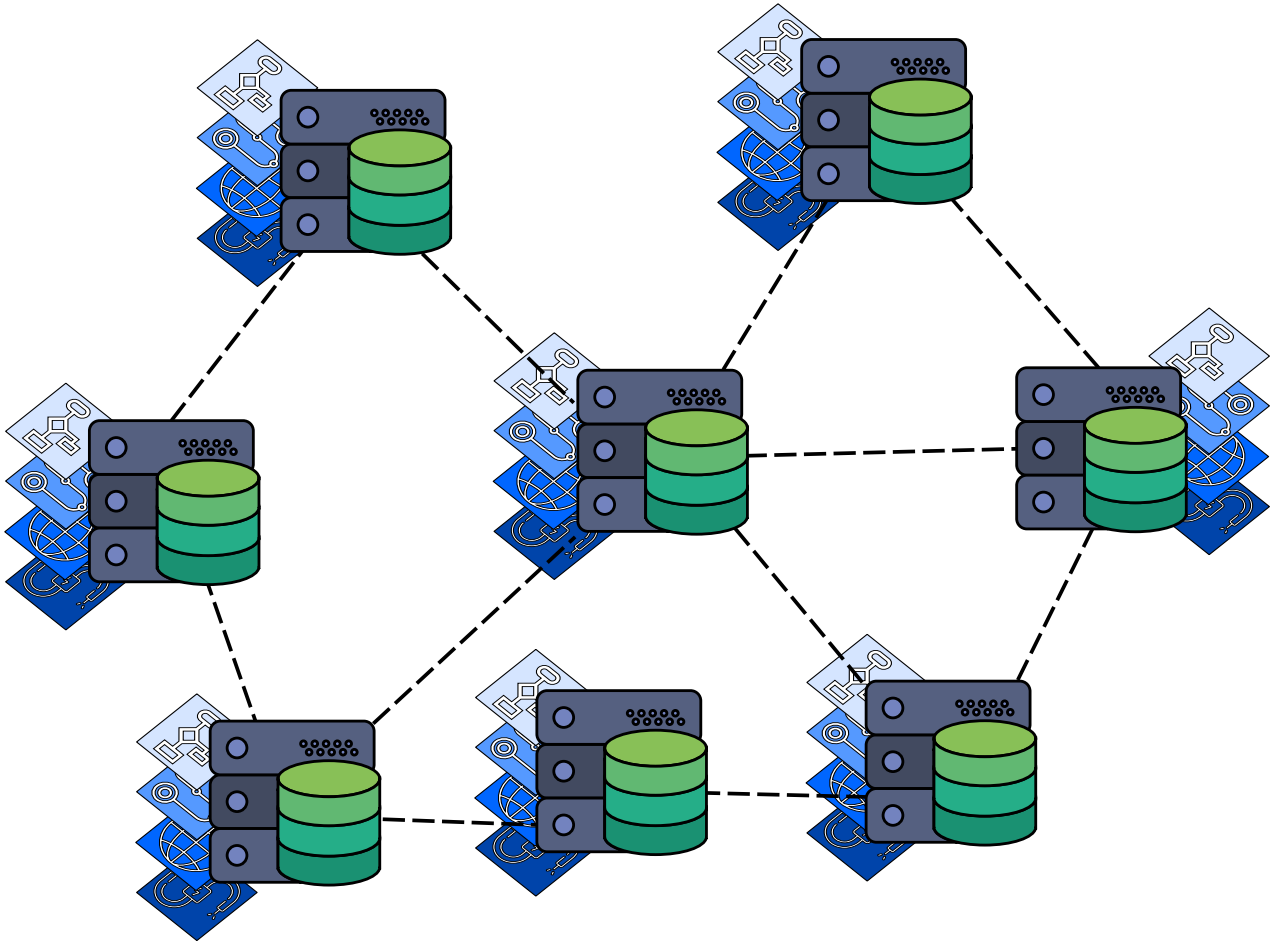
[a] State machine ≈ program

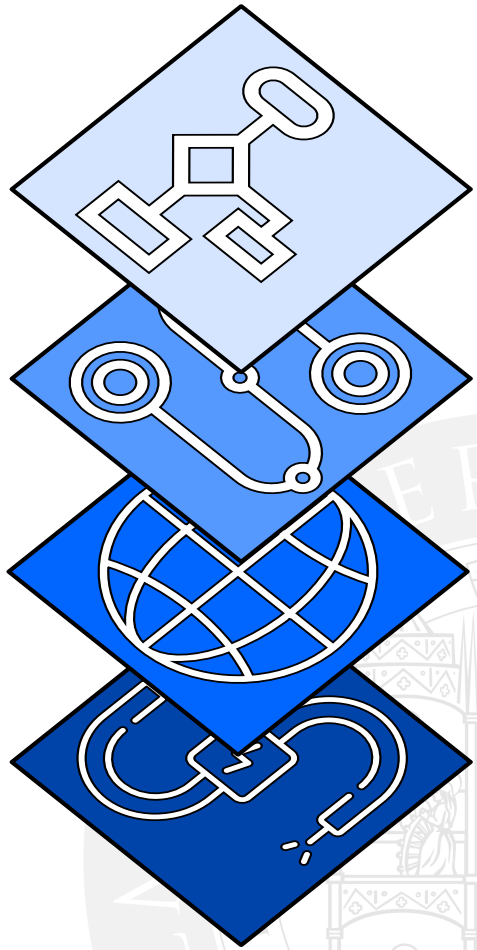
A network of replicas

When distributed, we say the processors constitute a peer-to-peer (P2P) network of replicas, all exposing the same observable behaviour

! no assumption about the topology

State Machine Replication (SMR)





Deterministic state**less** computations

$$\textit{Input} \longrightarrow \textit{Computation} \longrightarrow \textit{Output}$$

The computation is deterministic if it *always* produces the same output when it is performed on the same input.

- Can be arbitrarily replicated
- Replicas can be run in parallel

Deterministic

```
class Calculator {
  int sum(int x, int y) {
    return x + y;
  }
}
```

Non-deterministic

```
class Lottery {
  int extract(int max) {
    Random fate = new Random();
    return fate.nextInt(max);
  }
}
```

Deterministic state**ful** computations I

$$(Input, State) \longrightarrow Computation \longrightarrow (Output, State')$$

The computation is deterministic if it *always* produces the same output when it is performed on the same input and state.

- Can be arbitrarily replicated & replicas run in parallel
- All replicas must be initialised within the same initial state
- All inputs must be submitted to all replicas in the same order[1]
  - They all move through the same sequence of states
  - Maintaining the consistency of the state among on inputs

[1] input $\approx$ method call

Deterministic state**ful** computations II

Deterministic

```
class Ledger {
  Map<String, Integer> balances = // all accounts to 0

  void deposit(String userID, int money) {
    balances[userID] += money;
  }

  boolean transfer(String sender, String receiver, int money) {
    if (balances[sender] >= money) {
      balances[sender]   -= money;
      balances[receiver] += money;
      return true;
    }
    return false;
  }
}
```

Deterministic state**ful** computations III

Non-deterministic

```
class RaceCondition {
  int shared = 0;
  Thread t1 = new Thread( () -> shared++ );
  Thread t2 = new Thread( () -> shared-- );

  int guess() {
    t1.start();  t2.start();  t1.join();  t2.join();
    return shared;
  }
}
```

The Blockchain is essentially a means for replicating deterministic stateful computations

"Universal" State Machine Replication

$$UTM : TM = Interpreter : Program = SMR : ?$$

! UTM $\stackrel{def}{=}$ Universal Turing Machine — TM $\stackrel{def}{=}$ Turing Machine

- We can replicate a stateful deterministic program implementing a particular business logic

! e.g. a bank ledger

- In the exact same way, we could replicate a deterministic program implementing an interpreter

! interpreter $\approx$ a program which executes other programs

- The API of such an interpreter would enable programs to be deployed, undeployed, invoked, etc.

Smart-contracts-enabled Blockchains essentially act as replicated "universal" state machines on which smart contracts (SC) can be deployed

! Smart contract $\approx$ a program deployed on such a machine

"Universal" State Machine Replication – Example

```
class VirtualMachine {
  Map<String, Program> processes = // empty
  Compiler cc = // ...

  void deploy(String pid, String code) {
    Program newProgram = cc.compile(code);
    processes[pid] += newProgram;
  }

  Object invoke(String pid, Object[] args) {
    Object result = null;
    if (processes.containsKey(pid)) {
      result = processes[pid].call(args);
    }
    return result;
  }
}
```

SMR and **Distributed** Systems

- Messages may be lost, reordered, or duplicated by the network[2]
  - each node may perceive a different view about the system events
- Lack of global time

  $\Longrightarrow$ lack of total ordering of events

  $\Longrightarrow$ lack of trivial consistency
- Consistency, Availability, Partition-resistance (CAP) theorem [7]

  $\Longrightarrow$ you cannot have more than two of them
- Authentication is required if the replicated service is user-specific

[2] messages $\approx$ inputs to replicated processes

SMR, Middleware, and Consensus I

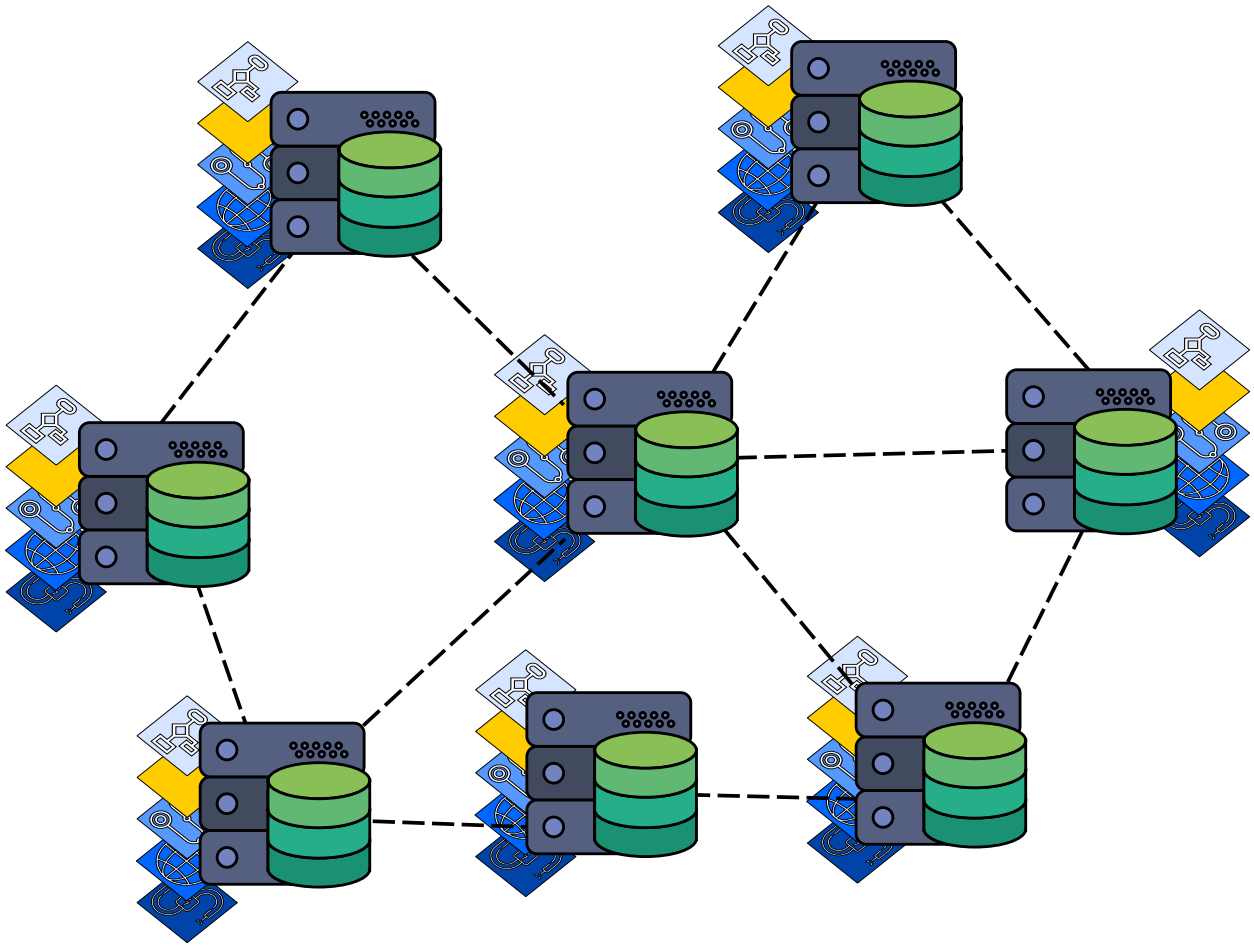
Each replica is executed on top of a middleware taking care of validating & ordering inputs for the replicated program

- It is then invoked on all nodes with the same sequence of inputs
- The middleware makes nodes participate to a consensus protocol
  - i.e. a distributed algorithm aimed at selecting the *next* input
  - ... producing the so-called atomic broadcast

  ! Fischer, Lynch and Patterson (FLP) theorem [15]

  $\Longrightarrow$ impossibility of consensus without timing assuptions

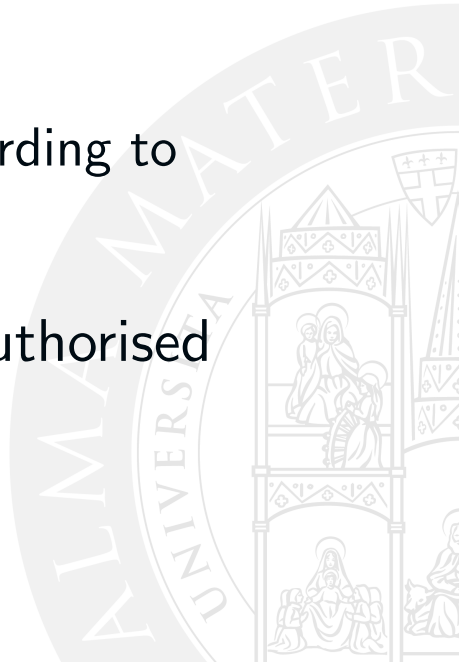
SMR, **Middleware**, and Consensus II



SMR and **Open** Distributed Systems

- How can we *prevent* a protocol participant from
  - lying w.r.t. the protocol rules or exchanged data?
  - being buggy, therefore breaking the rules or producing wrong data?
  - crashing?
  - . . . in general: being byzantine?
- Long story short: we can't.
- BUT we can tolerate *some* byzantine nodes
  - ! **Less than 1/3 of the total amount of nodes**, according to Lamport's Byzantine Generals Problem solution [19]
- We can also ease the recognition of prohibited or unauthorised behaviours by employing cryptography

  e.g. Pub/Priv key pairs for user authentications
  e.g. 1-Way Hash functions and MAC for data integrity



SMR and Open Distributed Systems

Takeaway

The blockchain is a smart way to achieve (U)SMR, dealing with – i.e., mitigating – well known issues of open distributed systems.

Disclaimer

Most of Blockchain-related works describe a *specific* blockchain technology (BCT henceforth) using a bottom-up approach. I believe this approach hinders generality and limits the discussion about what we can do on top of BCTs. In this section, I try to present the blockchain in a top-down way, synthesising informations from a number of sources, being [23], [28], [3] the most prominent ones. Errors and misunderstanding are possible, and in any case they are my sole responsibility.

The following description of the blockchain architecture and functioning is strongly inspired to Ethereum[3], being the most mature, studied, and documented smart-contracts-enabled BCT.

— G. Ciatto

[3] https://github.com/ethereum/wiki/wiki/White-Paper

Overview

Blockchain Technology (BCT)

A clever implementation of a SMR system keeping track of which users own some assets (representations), by means of a replicated ledger

e.g. The `Ledger` snippet

Smart-contracts-enabled BCT

A clever implementation of a USMR system keeping track of assets (representations) owned by entities – there including smart-contracts (SC), i.e. processes, owning code and state –, by means of a replicated ledger

e.g. The `VirtualMachine` snippet

Entity identifiers

Users

- Users are supposed to own (at least) one (K_{pub}, K_{pr}) key pair
- They are identified by some function $f(K_{pub})$ of their public key
  - e.g. 1-way-hash functions
  - e.g. digital certificates issued by some trusted CA

! Identifiers are also known as addresses in this context

Permission**ed** vs Permission**less**

- Either each user owns multiple non-intelligible identifiers...
  - ✓ Pseudonymity
  - ✓ Decentralised
  - × Sybil-attack [12]
- ...or he/she owns a single certified identifier
  - × Single point of failure/trust

! Smart-contracts-enabled BCTs identify both smart contracts' instances and users by means of the same sort of addresses

The world state I

The system state

$$\begin{array}{rcl} \langle \mathit{SystemState} \rangle & ::= & \texttt{entityID} \mapsto \langle \mathit{Account} \rangle \\ & | & \langle \mathit{SystemState} \rangle \ \langle \mathit{SystemState} \rangle \end{array}$$

The system state *conceptually* consists of a mapping between entity identifiers and *arbitrary* account data related to that account

The world state II

The account state

e.g. $\langle Account \rangle ::= (\texttt{balance}, \langle Storage \rangle, \langle Code \rangle, \langle Metadata \rangle)$

The account state *conceptually* consists of several fields keeping track of what a particular entity currently owns. The fields may vary depending on

- The blockchain nature
- Whether the entity is a smart contract or a user

e.g. BCTs coming with native cryptocurrencies, usually keep track of accounts `balances` (at least)

e.g. Smart-contracts-enabled BCTs, may keep track of their source code and internal $\langle Storage \rangle$

The world state III

```
class Ledger {
  Map<String, Integer> balances = ...
  //                   ^^^^^^^^
  //                system state

  void deposit(String userID, int money) {
    //                ^^^^^^
    //           entity identifier

    balances[userID] += ...
    //^^^^^^^^^^^^^^
    //account state
  }

}
```

Several possible implementations

- Unspent Transaction Output (UTXO)
  e.g. Bitcoin [23]
- Account-based
  e.g. Ethereum [28]
- Key-value store
  e.g. Hyperledger Fabric [3]

Transactions (a.k.a. inputs or messages) I

(Informal) Definition

$\langle \textit{Transaction} \rangle ::= (\texttt{txID}, \texttt{issuerID}, \langle \textit{Signature} \rangle, \texttt{recipientID}, \texttt{value}, \langle \textit{Data} \rangle)$

Transactions encode (world) state variations yet to be performed.

`txID` — the transaction progressive number

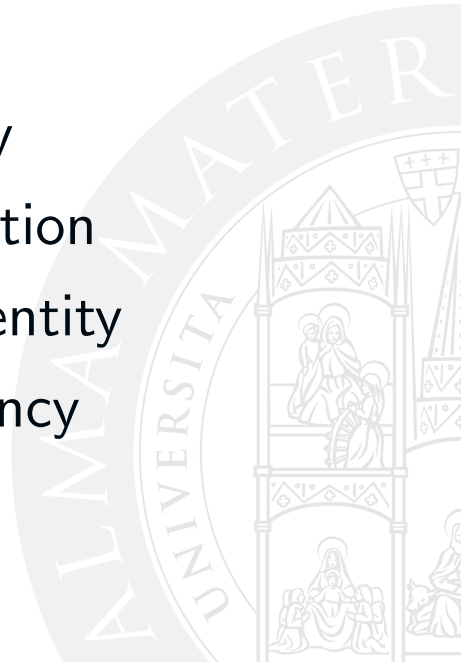
`issuerID` — the address of the transaction issuer entity

$\langle \textit{Signature} \rangle$ — the cryptographic signature of the transaction

`recipientID` — the address of the transaction recipient entity

`value` — some non negative amount of cryptocurrency

$\langle \textit{Data} \rangle$ — some arbitrary data



Transactions (a.k.a. inputs or messages) II

Transaction use cases (for smart-contracts-enabled BCTs)

Deployment TX — if $\texttt{recipientID} = \varnothing \wedge \langle \mathit{Data} \rangle = \texttt{code}$

Invocation TX — if $\texttt{recipientID} = \texttt{address} \wedge \langle \mathit{Data} \rangle = \texttt{whatever}$

Money transfer TX — if $\texttt{recipientID} \neq \varnothing \wedge \texttt{value} > 0 \wedge \langle \mathit{Data} \rangle = \varepsilon$

Transactions life-cycle — part 1

1. A issuer user compiles a transaction
2. He/she signs it with his/her private key K_{pr}
3. His/her node spreads the transaction over the P2P network
4. Peers only take into account valid transactions
5. ...

Transactions (a.k.a. inputs or messages) III

Transactions validity

In order for a transaction to avoid being dropped by peers:

- it must be well formed
- the signature must match the issuer address
- the signature must certify the transaction integrity
- the issuer's balance must be $\geq$ `value`

! Even once a valid transaction has been spreaded over the network, there is no guarantee on when it will be executed

Blocks and block chains I

(Informal) Definition

$$\begin{array}{rcl} \langle \mathit{Block} \rangle & ::= & (\texttt{prevHash}, \texttt{index}, \texttt{time}, \langle \mathit{TxList} \rangle, \langle \mathit{FinalState} \rangle) \\ \langle \mathit{TxList} \rangle & ::= & (\langle \mathit{Transaction} \rangle, \langle \mathit{IntermediateState} \rangle) \\ & | & \langle \mathit{TxList} \rangle \langle \mathit{TxList} \rangle \end{array}$$

Blocks are timestamped, hash-linked lists of transactions.

`prevHash` — the hash of the previous block

`index` — the index of the current block

`time` — the timestamp of the current block

$\langle \mathit{TxList} \rangle$ — the list of transactions included into the current block and the intermediate system states they produces

$\langle \mathit{FinalState} \rangle$ — the system state resulting from applying all transactions in $\langle \mathit{TxList} \rangle$, respecting their order

Blocks and block chains II

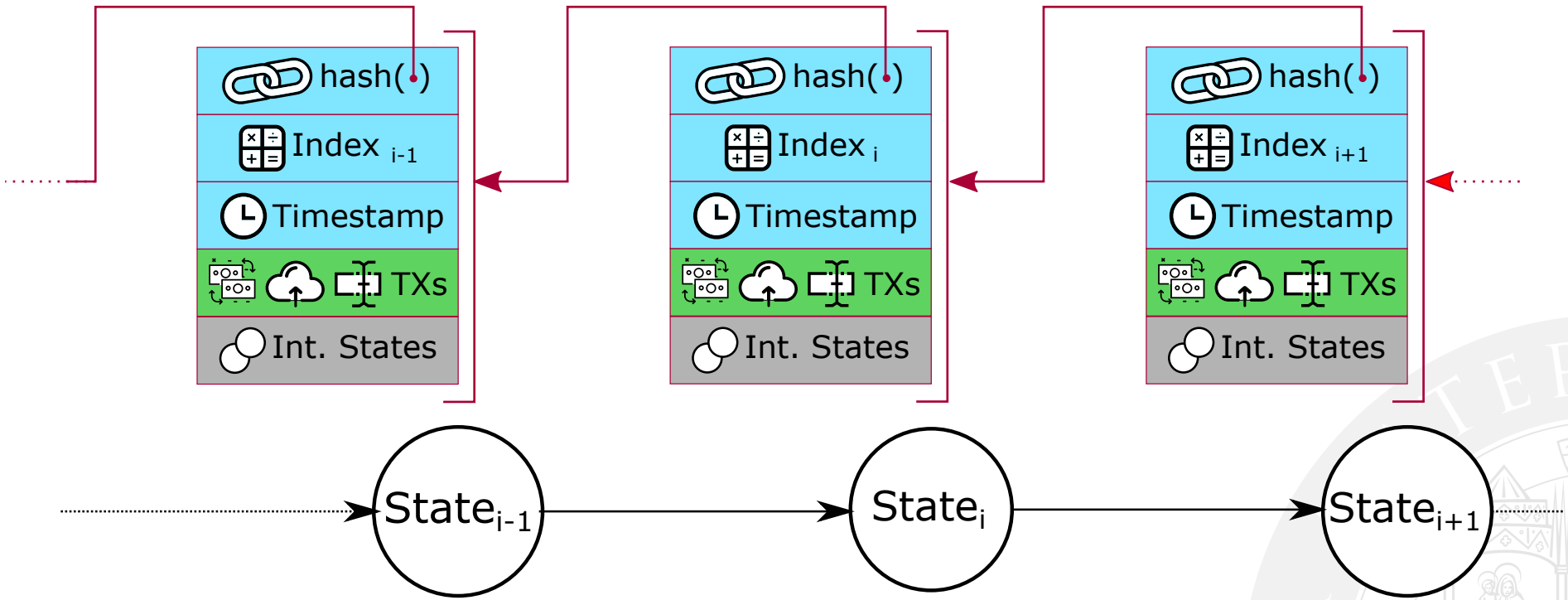


Figure: Graphical representation of the Block-chain from the global p.o.v.

Blocks features

- Replication + Hash-chaining $\rightsquigarrow$ Untamperability of the past
- Hash-chain + Time + Ordering $\rightsquigarrow$ Timestamping/notary service [16]
- Hash-chain + Crypt. signatures $\rightsquigarrow$ $\begin{cases} \text{Accountability} \\ \text{Non-repudiation} \end{cases}$
- They are supposed to be published (almost) periodically

! In the general case $\lim_{n\to\infty} \mathbb{P}[inconsistent(B_i)] = 0$, where
  - B_i is the i^{th} block
  - n is the amount of successor blocks of B_i
  - $inconsistent(B_i)$ is true if not all nodes agree on the content of B_i

A block's life I

The **genesis** block

The very first block is assumed to be shared between the initial nodes

Blocks life-cycle — part 1

Each node, *periodically*:

1. listens for transactions published by other nodes
2. validates, consistency-checks & executes them
3. compiles the new local candidate block
4. participates to the consensus algorithm

   i.e. negotiates the next block to be appended to the blockchain

   ! this phase include a spreading of the block to peers

A block's life II

Transactions life-cycle — part 2

4. the transaction is validated by peers upon reception
   - and dropped if invalid
5. each transaction is eventually executed
   - producing an intermediary state
6. they are both included into some block
7. the block is eventually appended to the blockchain

   i.e. a consensus protocol confirms the block
   - (there including its transactions)

! **These life-cycles may vary a lot depending on the specific consensus algorithm employed**

The network point of view

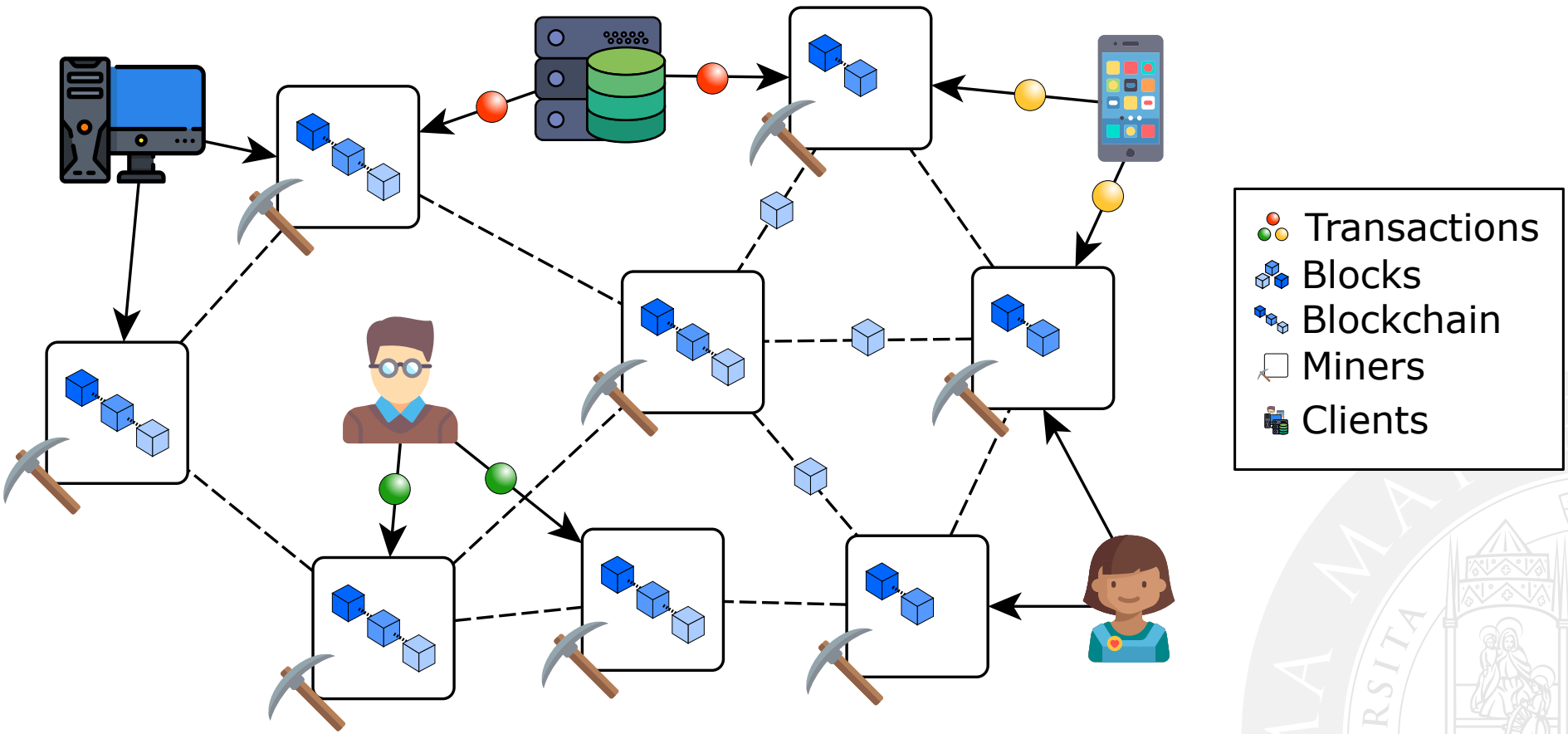


Figure: Graphical representation of the Block-chain from the network p.o.v.

Consensus & Mining I

Permission-**ed** BCTs

Constrain users' IDs through CAs

$\Downarrow$

"Classical" quorum/leader-based consensus algorithms

- BFT algorithms
  - e.g. PBFT [9]
  - e.g. BFT-SMaRt [26]
  - e.g. HoneyBadger BFT [22]
- Non-BFT algorithms
  - e.g. Paxos [17]
  - e.g. Raft [18]
  - e.g. ZooKeeper, Google Chubby

Permission-**less** BCTs

Open access to any (K_{pub}, K_{pr})

$\Downarrow$

"Novel" competition-based approaches

- e.g. Proof-of-Work [6]
- e.g. Proof-of-Stake [1]
- e.g. Proof-of-Elapsed-Time [11]
- e.g. IOTA Tangle [2]

Comparisons & surveys in [5, 8]

Consensus & Mining II

Permission-**ed** BCTs

"Classical" quorum/leader-based consensus algorithms

- Assumptions on the amount N of nodes
  UB up to $\sim$ 100 / 1000
- High throughput in terms of TXs/second
  OoM $\sim$ 1000 TXs/s
- "Exact" consistency
- Ideal for closed multi-administrative organizations

Permission-**less** BCTs

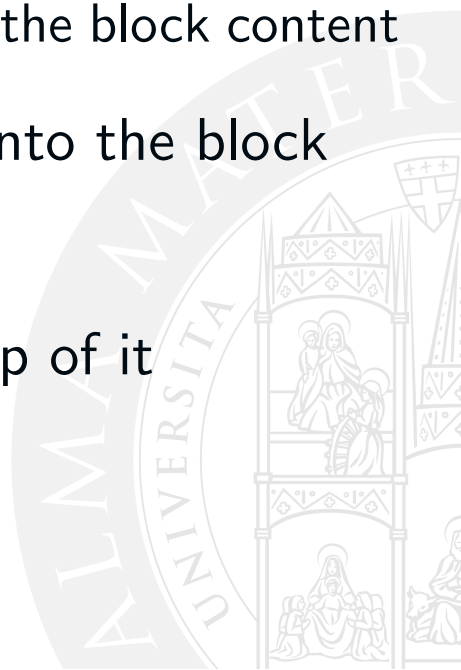
"Novel" competition-based approaches

- No assumption on N
  UB virtually ∞
- Low throughput
  OoM $\sim$ 10 TXs/s
- Probabilistic consistency
- Ideal for open systems

Proof-of-Work (PoW)

PoW (a.k.a. mining) — the typical approach in cryptocurrencies

- Nodes (a.k.a. miners) compete to be the first one to solve a computational puzzle, once every ΔT seconds
  - ₿ finding a block hash having a given amount of leading zeros
  - Ξ hashing (pseudo)random pieced of data attained form the block content
- The proof of the effort is easy to verify and included into the block
- The block is spreaded on the P2P network
- Other miners confirm the novel block by mining on top of it
- Forks (i.e. inconsistency) are eventually aborted
  - ₿ Longest comulative difficulty
  - Ξ Greedy Heaviest Observed SubTree (GHOST [25])



PoW interesting features

- Competition-based, local, eventually-consistent, stochastic approach
- Self-adaptive mining difficulty, s.t. $\mathbb{E}[\Delta T] = const$
  - ! the system update frequency is $1/\mathbb{E}[\Delta T]$
- Only computing power (CP) matters here
  - Sybil-attack resistant
  - CP distribution & Majority rule (51% attack) [14]

! Endows the cryptocurrency with its economical value

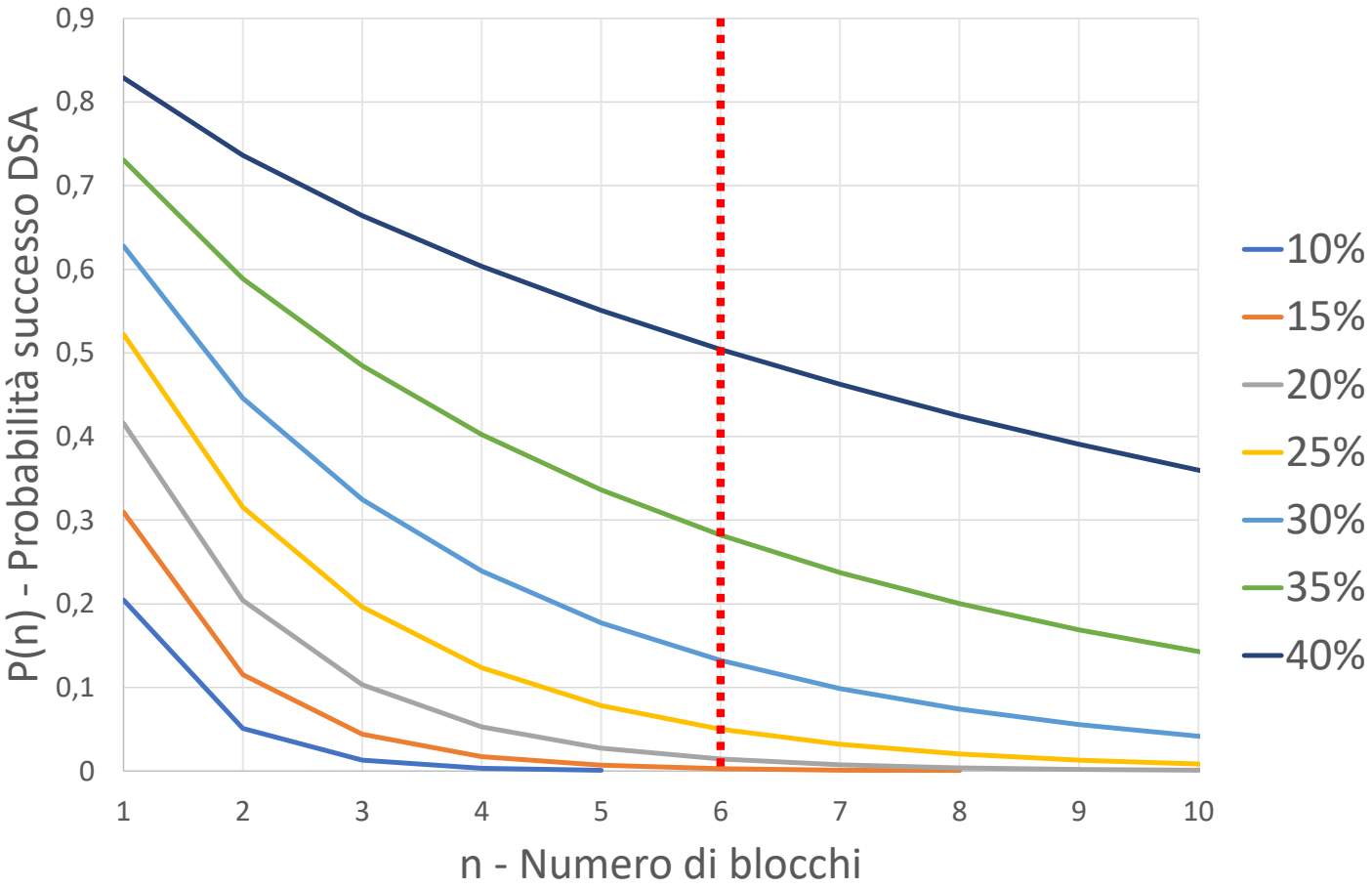
! Miners require economical compensation for their effort

PoW security

$$r = \frac{adversary_{CP}}{honests_{CP}}$$

$$\mathbb{P}[n \mid r] = 1 - \sum_{k=0}^{n} \frac{(nr)^k e^{-nr}}{k!}(1 - r^{n-k})$$

(see [23])



In Bitcoin:

- $n_{threshold} = 6$
- $\approx 1h$ since $\mathbb{E}[\Delta T] = 10m$
- 99.999% secure if $adversary_{CP} < 13\%\ total_{CP}$

Smart contracts [27]

(Informal) Definition

Stateful, *reactive*, *user-defined*, *immutable*, and deterministic processes executing some *arbitrary* computation *on the blockchain*, i.e., while being replicated over the blockchain network

Stateful — they encapsulate their own state, like OOP's objects

Reactive — they can only be triggered by issuing some invocation TX

User-defined — users can deploy their smart contracts implementing an arbitrary logic by issuing a deployment TX

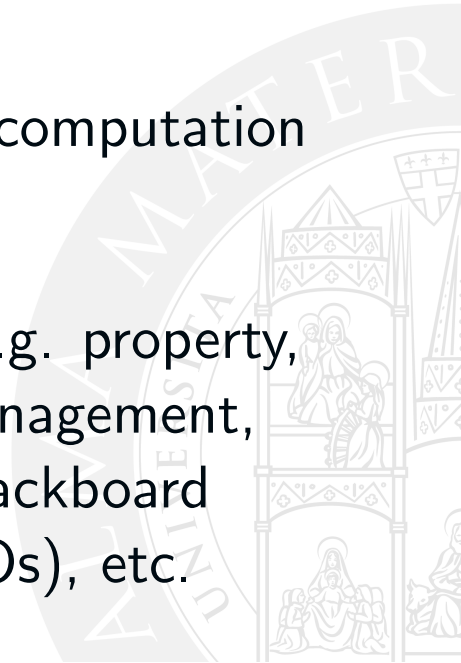
Immutable — their source/byte-code cannot be altered after deployment

Arbitrary — they are expressed with a Turing-complete language

Replicated — the blockchain is essentially a replicated interpreter, employing a consensus protocol to synchronise the many smart contracts replica

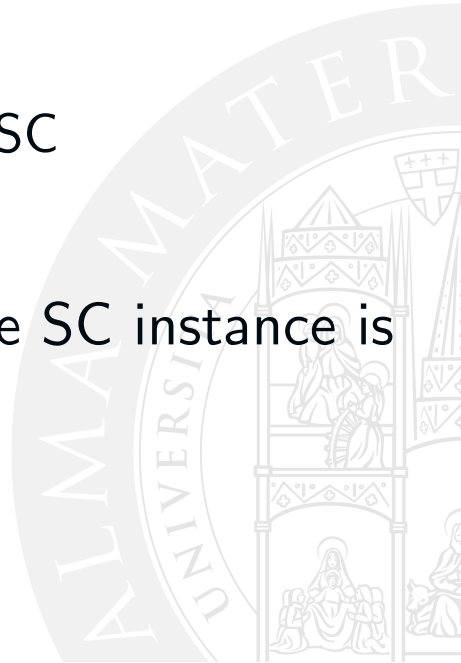
Smart contracts interesting features and expectations

- Immutability + Untamperable + Accountability + Decentralisation
  $\Longrightarrow$ can be trusted in handling financial operations between organizations
  - (even easier with native cryptocurrency)
- The code is always right, the true history is on the blokchain
  - reducing disputes
  - removing the need for arbitration
- Lack of situatedness: totally disembodied [21] data & computation
  - Like the cloud, but with no single point of control
- Killer applications: cryptocurrencies, asset-tracking (e.g. property, notary, medical-records, etc.), naming systems, ID management, access control, voting systems, reputation systems, blackboard systems, Distributed Autonomous Organizations (DAOs), etc.



Smart contracts deployment

1. Initially there exists no smart contract
   - i.e., the system state comprehend no smart contract entity
2. A user can instantiate a smart contract by publishing its source/byte-code within a deployment TX
   - the TX also initialises the SC state
   - the protocol assigns an univocal address to the novel SC
3. Once the transaction is confirmed, you can assume the SC instance is running on all nodes
   - no such a big effort: it is just listening for invocations



Smart contracts invocation

1. A user can trigger an already deployed SC by publishing an invocation TX
   - specifying the SC address as recipient
   - providing some input data specifying the requested computation
2. Eventually, each node will receive the TX and execute it
   - the SC code usually decides what to do given the provided input data
3. If the computation terminates without exceptions, any produced side effects (on the SC state) become part of the new intermediate system state
   - Otherwise, they are simply dropped, this the new intermediate state coincides with the previous one
4. The wrapping block is eventually confirmed by the consensus protocol, and the invoked computation can be actually considered executed

Issues arising from Turing-completeness

What would be the effect of invoking such a smart-contract?

```
class Bomb {
  int i = 0;

  void doSomething() {
    while (true) {
      i = (i + 1) % 10;
    }
  }
}
```

- BTCs cannot filter out non-terminating computation since termination is non-decidable
- In open systems, users cannot be assumed to simply well-behave

$\Longrightarrow$ Need to prevent/discourage users from deploying/invoking infinite or long computations

Ethereum and Gas

Ethereum proposes gas, i.e., making users pay for computation executions:

- TXs are endowed with two more fields: `gasLimit` & `gasPrice`
  - Miners could delay TXs having a low `gasPrice`
  - Users can increase their priority by increasing `gasPrice`
- Upon execution, each bytecode instruction increases the g counter
  - according to a price list defined in [28]
- Whenever $g >$ `gasLimit` an exception is raised, reverting side effects
- In any case, upon termination, the issuer balance is decreased of $\Delta ETH =$ `gasPrice` $\cdot\, g$
  - The winning miner can redeem ΔETH as a compensation for its computational effort

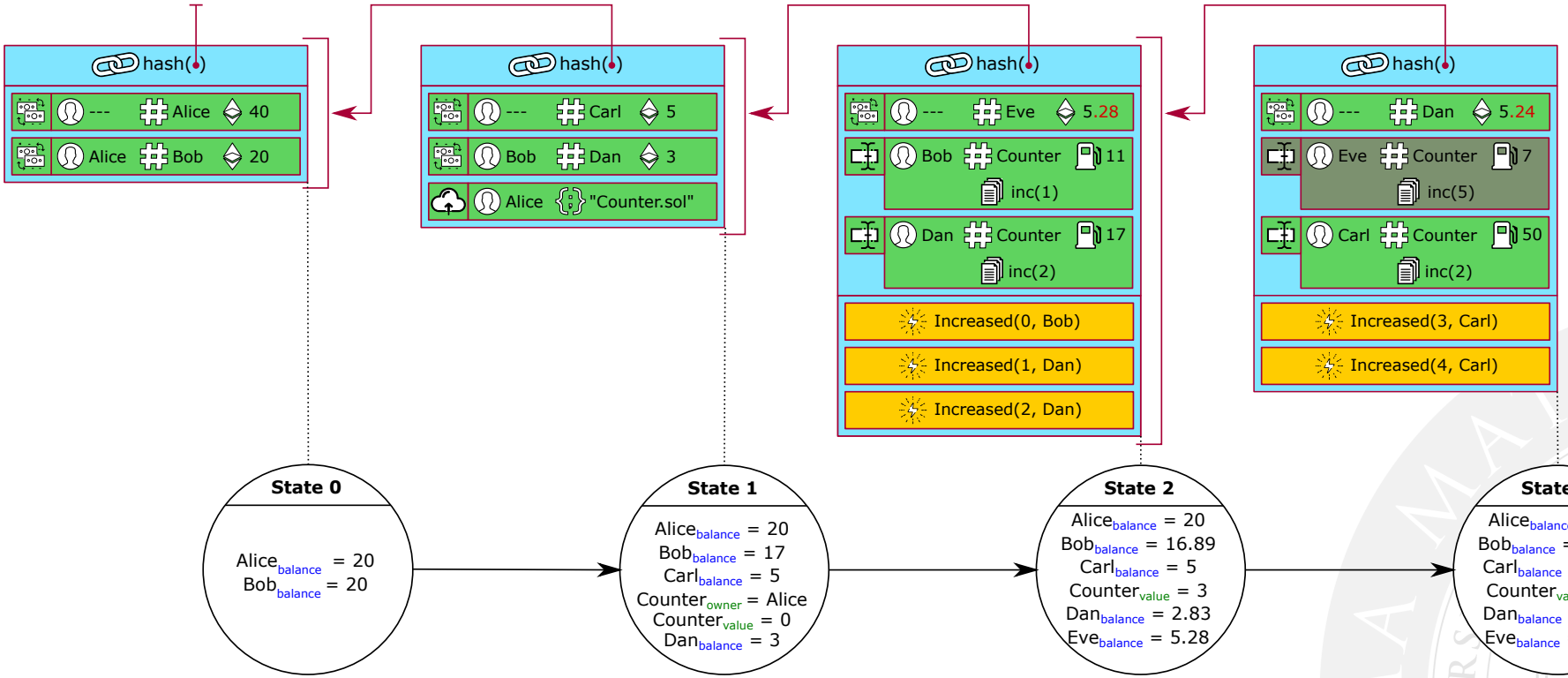
The economical dimension of computation has to be taken into account when designing Ethereum smart contracts

Ethereum smart contract example with Solidity[4]

```
contract Counter {

  event Increased(uint oldValue, address cause);

  address owner;          uint value;

  function Counter() public { owner = msg.sender; } // <-- constructor

  function inc(uint times) public { // <-- API
    for (uint i = 0; i < times; i++) {
      emit Increased(value++, msg.sender);
    }
  }

  function kill() public { // <-- API
    require(msg.sender == owner);
    suicide(owner);
  }

  function () { // <-- fallback
    throw;
  }

}
```

[4] https://solidity.readthedocs.io

Ethereum smart contract example with Solidity



Smart contracts issues I

No privacy or secrets

Every information ever published on the blockchain stays on the blockchain

- The private state of a smart contract is not secret
- Pseudo-anonymity can be broken with statistics & data-fusion
- Illegal/anti-ethic behaviour can be revealed years later

! No secret voting?!

Smart contracts issues II

Poor randomness

It is difficult to achieve (pseudo-)randomness because of the lack of trustable sources

- Real randomness cannot be employed (replicas would diverge)
- Most of the blocks observable information are under the control of the miner

  e.g. timestamp, height, hash, etc.
- The block hash seems a good choice
  - but this is an egg-and-chicken problem

! No lottery?!

Smart contracts issues III

Smart contract inter-communication

Can a SC interact with another one? Which is the exact semantics of doing so? Is OOP the best programming paradigm?

In Ethereum, SC are essentially objects communicating by means of synchronous method calls. The callee SC are referenced by callers by means of their address:

- the control flow originating from a user may traverse more than a SC
- the caller waits for the callee
- unattended re-entrancy if difficult to avoid [4, 20]
- and it may lead to undesired behavioural subtleties and frauds [13]

https://medium.com/gus_tavo_guim/reentrancy-attack-on-smart-contracts

Smart contracts issues IV

Impossibility to fix bugs

SC code is immutable. Immutability is both a blessing and a curse. Buggy contracts cannot be fixed, updated, replaced, or un-deployed

- Buggy, misbehaving, fraudolent SCs will remain so, wasting miners resources
- Paramount importance of correct-design and formal validation
  - a problem *per se* in Turing-complete languages
- Behavioural OOP design patterns are possible, but critical because of the previous issue

Smart contracts issues V

Lack of proactiveness

SCs are purely reactive computational entities

- They always need to borrow some user's control flow
- They are time-aware but not reactive to time
- They cannot schedule or postpone computations
  - no periodic computation (e.g. payment)

Smart contracts issues VI

Disembodiement [21] & lack of concurrency

Computation is logically located everywhere and transactions are strictly sequential. This may be a wasteful approach in some cases:

- Independent computation cannot be executed concurrently
- Computations only making sense locally need to be replicated globally
- Heavy computations cannot be splitted into parts to be run concurrently

Smart contracts issues VII

Granularity of computation-related costs

Ethereum is not the first platform applying a price to computation:

e.g. Common practice on the Cloud, and the X-as-a-Service paradigm

BTW, is the instruction-level cost granularity the better one?

e.g. In the most trivial implementation of a publish-subscribe mechanism, it is the publisher paying the variable price

Programming paradigms for smart contracts

Problem

! HLL = High Level Language

SCs research care a lot about HLLs but some issues are related to their underlying operational semantics:

- Synchronous calls are usually hard coded by construction
- Poor care for what concerns inter-SC interaction
- Lack of control flow encapsulation
- Lack of proactiveness

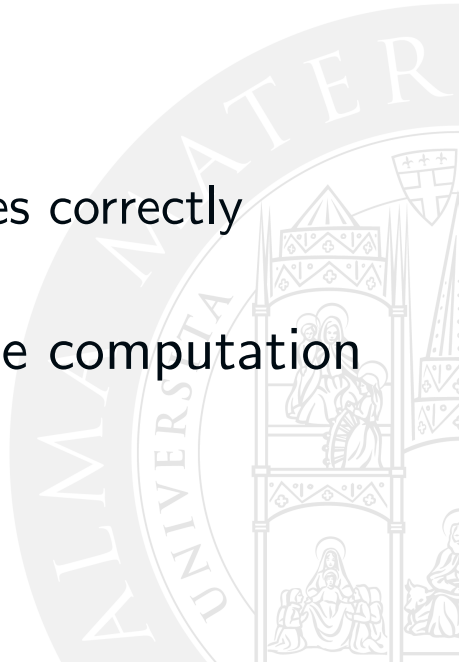
Goals

Investigating the adoption of interaction-friendly paradigms such as actors or agents

Smart contracts as Actors I

Possible modification to SCs operational semantics:

- Asynchronous message passing as the unique means of inter-SC communication + control flow encapsulation
  - The total ordering of events perfectly matches the event-loop semantics of Actors
- Sending a message implies issuing an invocation TX
  - Analogously to what users do
  - Messages are sent only after the current TX terminates correctly
- Selective/guarded receive for enabling or delaying some computation
- Private, synchronous call are still possible
  - Can be used to implement pure computations



Smart contracts as Actors II

Interesting questions arising from this vision:

? Who is paying for SC-initiated control flows?

? How to compensate miners for delayed computations?

Possible activities

- Re-thinking or editing some BCT formal semantics in terms of actors
- Forking some existing BCT project to inject the actors semantics
- Designing (and developing?) a novel BCT project exposing an actor-based SC abstraction

Smart contracts and Multi-Agent Systems (MAS) I

There seems to be more degrees of freedom here:

- Different, possibly overlapping, declination of the Agent notion:
  e.g. Believes-Desires-Intentions (BDI), Agents & Artifacts (A&A)
- Different possible mappings are for: { Agent, Environment, Artifact? }
- Which choices are the best ones and why?

Smart contracts and Multi-Agent Systems (MAS) II

For instance, let's image SCs as BDI agents:

- Then, what's the environment? What can an agent perceive?
- Is goal-oriented reasoning useful in this context?
  - Should a SC reason about how to execute its business logic?
- What about epistemic actions?
  - Should a SC ask for unknown informations to other (human?) agents?
- Do multiple intentions (i.e., multiple control flows) make sense?
  - Who is paying for them?
  - Who is in charge for executing them? Using which concurrency model?

Possible activities

- Re-thinking or editing some BCT formal semantics in terms of agents, environment, artifacts, etc.

Logic-based smart contracts I

Problem

! HLL = High Level Language

SCs currently lack:

- high level understandability in their HLLs
- observability of the deployed source code
- some degree of evolvability enabling them to be modified (or fixed)

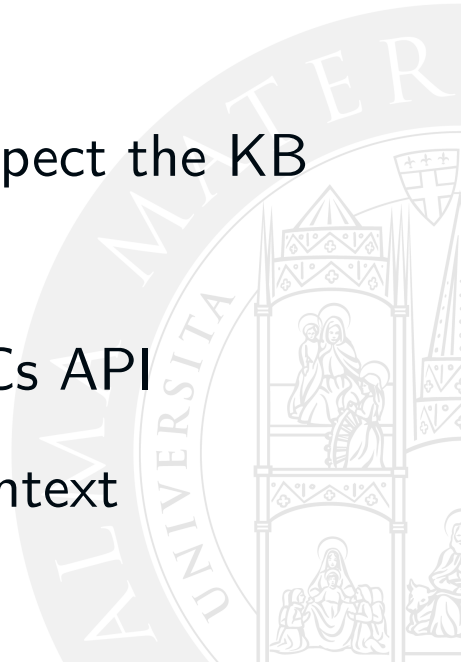
Goals

Investigating how the adoption of a logic interpreted language (e.g. Prolog) may improve SC for what concerns such aspects

Logic-based smart contracts II

Employing a logic language, such as Prolog, introduces some benefits:

- naturally declarative and goal-oriented, improving understandability
- static KB for the immutable code, dynamic KB for the mutable part
- `asserts` & `retracts` only affect the dynamic KB
  - thus enabling some sort of controlled evolvability
- being an interpreted language it always possible to inspect the KB
  - without disassemblers
- guarded/selective receive to enforce a boundary for SCs API
- context-aware predicates for inspecting the current context
  - similarly to Solidity's Globally Available Variables



Logic-based smart contracts III

. . . And some more questions:

- should the computational economic cost model be re-designed to embrace LP basic mechanisms?

! LP = Logic Programming

- how should logic SCs interact?

Possible activities

- Re-thinking or editing some BCT formal semantics to embrace such a vision
- Designing (and develop) such a novel vision from scratch

Blackboard-based approaches and smart contracts

Opportunity

Shared blackboards systems may take real advantage of the replication and fault-tolerance features they would inherit if deployed on top of a BCT layer. For instance:

e.g. tuple-based coordination

e.g. distributed logic programming

Goals

- Investigating whether BCTs are useful in such contexts or not.
- Considering such contexts as applications, looking for improvements to the BCTs

Tuple-based coordination on the Blockchain I

Can we build the archetypal LINDA model on top of BCTs?

- If yes, tuple spaces would inherit a lot of desirable properties
  - e.g. Decentralisation & replication, fault-tolerance, consistency, etc.

- ? Which computational economical cost model for LINDA primitives?
- ? How to handle control flow-related aspects?
  - e.g. suspensive semantics
- ? Can we inject programmability too?

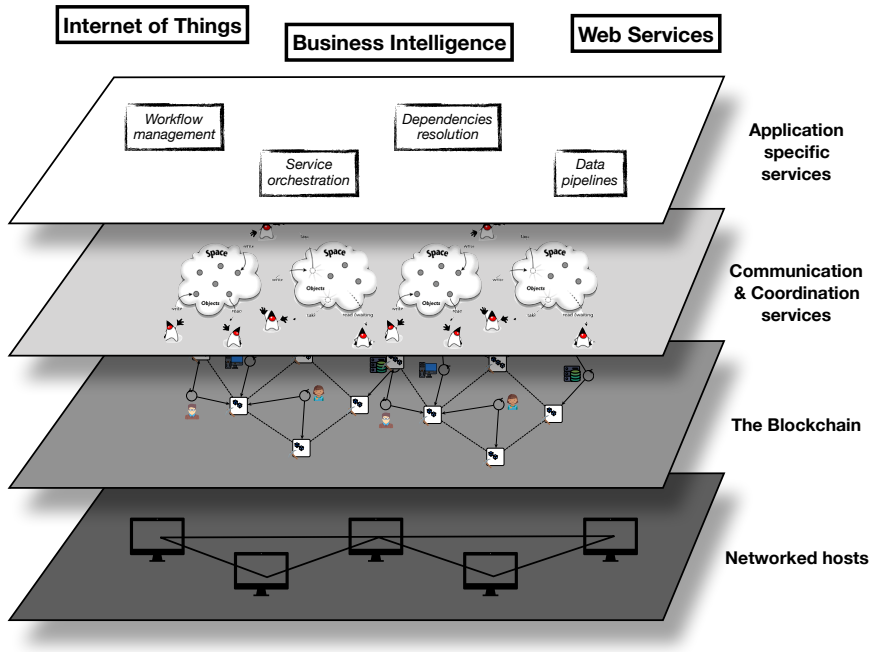


Figure: Our vision: BCTs as the backbone on top of which communication and coordination services are built

Tuple-based coordination on the Blockchain II

Possible activities

- Compare several BCTs from the coordination-capabilities point of view, modelling and implementing LINDA on top of them
- Compare several BCTs from the coordination-capabilities point of view, modelling and implementing ReSpecT on top of them

Distributed LP on the Blockchain

Can we employ the blockchain as a blackboard enabling distributed agents to cooperatively participate to some SLD reasoning process?

- Again, desirable properties would be "automatically" inherited
- LP-friendly economical incentives/disincentives could be conceived stimulating miners to adopt a particular strategy when building/exploring some proof-tree
- Concurrent LP has some well-known critic aspects
  - e.g. AND-parallelism, OR-parallelism, termination, non-termination, shared variables
- ? How to handle KB mutability while reasoning?

Possible activities

- Develop (at least) a proof of concept or sketched implementation showing the feasibility of concurrent, blockchain-based, SLD resolution process

Formal (meta-)model for BCTs and smart contracts

Problem

A part from Ethereum, other mainstream BCTs lack a formal semantics specification. Furthermore, a general meta-model comprehending them all is still missing.

Goals

- Defining a meta-model explaining all (or most) existing BCTs
  - or proving it to be impossible
- Defining an operational semantics for all (or most) existing BCTs
- Showing why the operational semantics of each BCT is an instance of the general meta-model

Possible activities

- SLR about the formal semantics of one or more BCTs
- Define your own formal semantics/meta-model

Simulating the blockchain

Problem

Some local consensus approaches lack formal theorems proving their properties or their sensibility to the parameters variation

e.g. ΔT, CP distribution, economical cost model, etc.

Goals

Designing & developing an agent-based simulation framework where such interrelated aspects can be studied *in silico*

Possible activities

- Develop the simulation framework and show its effectiveness by simulating a simple consensus model
- Design a complex consensus model to be simulated on the aforementioned framework to reveal critical parameters regions

Local consensus mechanisms

Problem

Classical BFT consensus algorithms are very powerful but their performance essentially degrades with the amount of nodes

Goals

Conceive, design, implement, and assess other local (stochastic?) consensus mechanisms ensuring some (possibly provable) security properties.

Possible activities

- SLR on classical/novel consensus mechanisms: compare & classify
- Implement some classical/novel consensus protocol
- Design your own (non-trivial) consensus mechanism

Concurrency, sharding & DAGs

Problem

BCTs lack real concurrency or situatedness (of both data and computations) and these lacks are inherited by SCs This is essentially a waste of storage/computational resources

Goals

Conceive a non-trivial solution enabling some of the following features:

- concurrent execution of independent SCs
- data and computation partitioning on different nodes
- branching/merging of the blockchain (making it a DAG)

Possible activities

- SLR on such aspects
- Design your own (non-trivial) concurrent BCT

Privacy & confidentiality for smart contracts

Problem

SCs lack confidentiality when interacting with users, and some means to hide their private internal state

Goals

Developing a cryptographic schema aimed at injecting some degree of confidentiality/privacy into smart contracts

Possible activities

- SLR on privacy/confidentiality-related aspects
- Design your own (non-trivial) cryptographic schema

References I

[1] Proof of stake.
https://en.bitcoin.it/wiki/Proof_of_Stake.

[2] The tangle.
https://iotatoken.com/IOTA_Whitepaper.pdf.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick.
Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains.
Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18, pages 1–15, jan 2018.

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli.
A survey of attacks on Ethereum smart contracts (SoK).
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 10204 LNCS(July):164–186, 2017.

References II

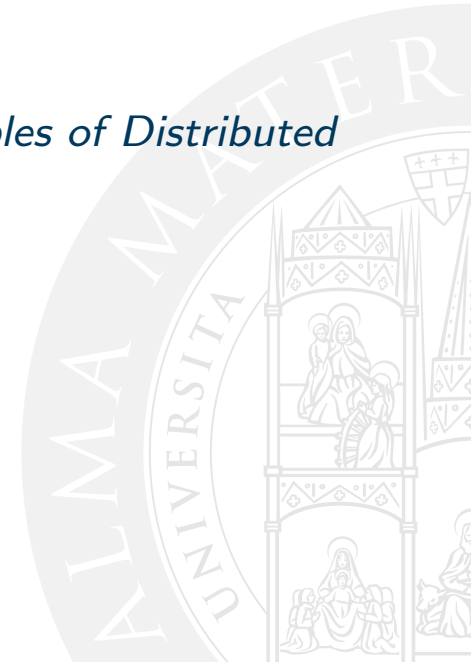
[5] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić.
The Next 700 BFT Protocols.
ACM Transactions on Computer Systems, 32(4):1–45, jan 2015.

[6] Adam Back.
Hashcash - A Denial of Service Counter-Measure.
Http://Www.Hashcash.Org/Papers/Hashcash.Pdf, (August):1–10, 2002.

[7] Eric A. Brewer.
Towards robust distributed systems (abstract).
In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[8] Christian Cachin and Marko Vukolić.
Blockchain Consensus Protocols in the Wild.
jul 2017.

[9] Miguel Castro and Barbara Liskov.
Practical byzantine fault tolerance and proactive recovery.
ACM Transactions on Computer Systems, 20(4):398–461, 2002.



References III

[10] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors.
Replication: Theory and Practice.
Springer-Verlag, Berlin, Heidelberg, 2010.

[11] Lin Chen, Lei Xu, Nolan Shah, Zhimin Gao, Yang Lu, and Weidong Shi.
On security analysis of proof-of-elapsed-time (poet).
In Paul Spirakis and Philippas Tsigas, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 282–297, Cham, 2017. Springer International Publishing.

[12] John R. Douceur.
The Sybil Attack.
pages 251–260, 2002.

[13] Quinn Dupont.
Experiments in Algorithmic Governance : A history and ethnography of " The DAO ," a failed Decentralized Autonomous Organization.
Bitcoin and Beyond, pages 1–18, 2017.

[14] Ittay Eyal and Emin Gün Sirer.
Majority is not enough: Bitcoin mining is vulnerable.
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8437:436–454, 2014.

References IV

[15] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson.
Impossibility of distributed consensus with one faulty process.
Journal of the ACM, 32(2):374–382, 1985.

[16] Stuart Haber and W.Scott Stornetta.
How to time-stamp a digital document.
Journal of Cryptology, 3(2):99–111, 1991.

[17] Leslie Lamport.
The Part-Time Parliament.
ACM Transactions on Computer Sys-tems, 16(2):133–169, 1998.

[18] Leslie Lamport, Benjamin C. Reed, Flavio P. Junqueira, Diego Ongaro, John Ousterhout, Michael a Olson, Keith Bostic, Margo Seltzer, Cynthia Dwork, Nancy Lynch, Larry Stockmeyer, Jim Shore, Fred B Schneider, Leslie Lamport, Miguel Castro, Barbara H Liskov, H.Zou, F.Jahanian, Leslie Lamport, Dahlia Malkhi, Lidong Zhou, X Zhang, D. Zagorodnov, M Hiltunen, Keith Marzullo, R.D. Schlichting, Navin Budhiraja, Keith Marzullo, Fred B Schneider, Sam Toueg, R. Al-Omari, Arun K. Somani, G. Manimaran, Flavio P. Junqueira, Benjamin C. Reed, Marco Serafini, Navin Budhiraja, Rachid Guerraoui, André Schiper, M. Pease, R. Shostak, Leslie Lamport, Dahlia Malkhi, Lidong Zhou, Lamport July, Barbara H Liskov, and James Cowling.
In Search of an Understandable Consensus Algorithm.

References V

Atc '14, 22(2):305–320, 2014.

[19] Leslie Lamport, Robert Shostak, and Marshall Pease.
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems, 4(3):382–401, 1982.

[20] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor.
Making Smart Contracts Smarter.
In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 254–269, New York, New York, USA, 2016. ACM Press.

[21] Stefano Mariani and Andrea Omicini.
TuCSoN on Cloud: An event-driven architecture for embodied/disembodied coordination.
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8286 LNCS(PART 2):285–294, 2013.

[22] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song.
The Honey Badger of BFT Protocols.

[23] Satoshi Nakamoto.
Bitcoin: A Peer-to-Peer Electronic Cash System.
Www.Bitcoin.Org, page 9, 2008.

References VI

[24] Fred B. Schneider.
Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial.
ACM Comput. Surv., 22(4):299–319, 1990.

[25] Yonatan Sompolinsky and A Zohar.
Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains.
IACR Cryptology ePrint Archive, 881:1–31, 2013.

[26] João Sousa and Alysson Bessani.
From Byzantine consensus to BFT state machine replication: A latency-optimal transformation.
Proceedings - 9th European Dependable Computing Conference, EDCC 2012, pages 37–48, 2012.

[27] Nick Szabo.
Smart Contracts: Building Blocks for Digital Markets.
Alamut.Com, (c):16, 1996.

[28] Gavin Wood.
Ethereum: a secure decentralised generalised transaction ledger.
Ethereum Project Yellow Paper, pages 1–32, 2014.

A gentle introduction to the Blockchain and Smart contracts

Giovanni Ciatto { giovanni.ciatto@unibo.it }

Talk @ Autonomous Systems Course, A.Y. 17/18
Dipartimento di Informatica, Scienza e Ingegneria—Università di Bologna

May 30, 2018